ICSC Qualification Round Solutions

Jaino C. Cabrera

August 23, 2025

Contents

Problem A: Neural Network Components

Statement

Identify and label the core components of a simple feed-forward neural network (multi-layer perceptron) used to predict customer satisfaction, based on the given figure.

Components and reasoning

$w_{21}^{(2)}$

Weight: from input$_2$ ("Tip received") to hidden neuron 1 (Box A, top). A weight scales the signal before it enters the next layer.

Σ

Summation: computes the pre-activation $z = \sum_i w_i a_i + b$, i.e. the weighted sum of incoming signals plus bias.

f

Activation function: transaforms the pre-activation z into the neuron's output $a = f(z)$ (e.g., sigmoid, ReLU, tanh), introducing nonlinearity.

○ Input neurons

The red circles correspond to features ("Duration stayed," "Tip received"), denoted x_1, x_2.

○ Bias node

A constant 1 connected to each neuron. Its learned weight acts as an additive term b, shifting the activation threshold.

○ Output neuron

The final prediction unit. Combines hidden activations into $\hat{y}$. For binary classification, typically passed through a sigmoid to yield $\hat{y} \in [0, 1]$.

Box A: Hidden layer

Intermediate neurons that transform inputs into learned features, not directly observed.

Box B: Output layer

Contains the single readout neuron that produces the network's decision.

$\hat{y}$ (prediction)

The network's estimate of customer satisfaction, $\hat{y} = f(z)$.

Relationship among the components

The computation in the output neuron follows

$$\underbrace{z}_{\Sigma\text{ block}} = \sum_i \underbrace{w_i}_{\text{weights}} a_i + \underbrace{b}_{\text{bias}}, \qquad \underbrace{\hat{y}}_{\text{output}} = \underbrace{f}_{\text{activation}}(z),$$

where a_i are the activations from Box A (hidden layer).

Thus, the forward pass is: inputs (red) $\rightarrow$ hidden layer (Box A) with weights + bias (blue, orange, brown) $\rightarrow$ activation (cyan) $\rightarrow$ output (green, Box B).

Problem B: Cake Calculator

Statement

Given two positive integers `flour` and `sugar`, compute:

1. the maximum number of cakes that can be produced, and
2. the remaining flour and sugar after producing that many cakes,

where each cake requires exactly 100 units of flour and 50 units of sugar.

Derivation and reasoning

Let k be the number of cakes produced (an integer, $k \geq 0$).
Each cake consumes 100 units of flour and 50 units of sugar, so k cakes consume

$$\text{flour used} = 100 \cdot k, \qquad \text{sugar used} = 50 \cdot k.$$

Feasibility requires:

$$100 \cdot k \leq \texttt{flour}, \qquad 50 \cdot k \leq \texttt{sugar}.$$

Solving for integer k gives:

$$k \leq \left\lfloor \frac{\texttt{flour}}{100} \right\rfloor, \qquad k \leq \left\lfloor \frac{\texttt{sugar}}{50} \right\rfloor.$$

Hence the maximal feasible k is

$$\texttt{cakes} = \min\left(\left\lfloor \frac{\texttt{flour}}{100} \right\rfloor, \left\lfloor \frac{\texttt{sugar}}{50} \right\rfloor \right).$$

Leftovers follow by subtraction:

$$\texttt{remaining\_flour} = \texttt{flour} - (100 \cdot \texttt{cakes}), \qquad \texttt{remaining\_sugar} = \texttt{sugar} - (50 \cdot \texttt{cakes}).$$

Implementation

Python — Cake Calculator

```python
import sys

def cake_calculator(flour: int, sugar: int) -> list:
    """
    Calculates the maximum number of cakes that can be made and the leftover
    ↪  ingredients.

    Args:
        flour: An integer larger than 0 specifying the amount of available flour.
        sugar: An integer larger than 0 specifying the amount of available sugar.

    Returns:
        A list of three integers:
        [0] the number of cakes that can be made
        [1] the amount of leftover flour
        [2] the amount of leftover sugar

    Raises:
        ValueError: If inputs flour or sugar are not positive.
    """

    if flour <= 0 or sugar <= 0: # Check if both values are positive
        raise ValueError("Flour and sugar must be positive integers.")

    cakes_flour = flour // 100 # Maximum number of cakes we can make with the given
    ↪ amount of flour
    cakes_sugar = sugar // 50 # Maximum number of cakes we can make with the given
    ↪ amount of sugar
    cakes = cakes_flour if cakes_flour < cakes_sugar else cakes_sugar # Smallest
    ↪ among the two limits the amount of cakes we can make

    return [cakes, flour - cakes * 100, sugar - cakes * 50] # [cakes we can make,
    ↪ leftover flour, leftover sugar]


# --- Main execution block. DO NOT MODIFY  ---
if __name__ == "__main__":
    try:
        # 1. Read input from stdin
        flour_str = input().strip()
        sugar_str = input().strip()

        # 2. Convert inputs to appropriate types
        flour = int(flour_str)
        sugar = int(sugar_str)

        # 3. Call the cake calculator function
        result = cake_calculator(flour, sugar)

        # 4. Print the result to stdout in the required format
        print(f"{result[0]} {result[1]} {result[2]}")

    except ValueError as e:
        # Handle errors during input conversion or validation
        print(f"Input Error or Validation Failed: {e}", file=sys.stderr)
        sys.exit(1)
    except EOFError:
        # Handle cases where not enough input lines were provided
        print("Error: Not enough input lines provided.", file=sys.stderr)
        sys.exit(1)
    except Exception as e:
        # Catch any other unexpected errors
        print(f"An unexpected error occurred: {e}", file=sys.stderr)
        sys.exit(1)
```

Complexity

All operations are constant-time integer arithmetic and comparisons. Hence

$$\text{Time complexity} = O(1), \qquad \text{Space complexity} = O(1).$$

Problem C: The School Messaging App

Given

The source alphabet and symbol probabilities:

Character	Probability	Character	Probability
A	0.20	G	0.05
B	0.15	H	0.05
C	0.12	I	0.04
D	0.10	J	0.03
E	0.08	K	0.02
F	0.06	L	0.10

Table 1: Character Probabilities

A Fano-produced prefix code for these symbols (given in the statement) is:

Character	Probability	Fano Code	Character	Probability	Fano Code
A	0.20	000	G	0.05	001
B	0.15	100	H	0.05	1011
C	0.12	010	I	0.04	0111
D	0.10	1100	J	0.03	1101
E	0.08	0110	K	0.02	1111
F	0.06	1010	L	0.10	1110

Table 2: Fano Codes for Characters

Question 1: Why variable-length codes help

Standard fixed-length encodings assign the same number of bits to every symbol. If there are M symbols, a fixed-length binary code uses $\lceil \log_2 M \rceil$ bits per symbol. This wastes bits when some symbols occur much more often than others.

Key idea: give short bit-strings to frequent symbols and longer bit-strings to rare symbols. The *average* number of bits per transmitted symbol becomes

$$\bar{L} = \sum_i p_i \ell_i,$$

where ℓ_i is the length of the codeword for symbol i. By choosing ℓ_i smaller for large p_i, $\bar{L}$ is reduced.

Concrete example. Suppose four symbols with probabilities

$$A : 0.70, \quad B : 0.10, \quad C : 0.10, \quad D : 0.10.$$

A fixed-length binary code must use $\lceil \log_2 4 \rceil = 2$ bits per symbol on average. One variable-length prefix code is:

$$A \mapsto 0 \text{ (length 1)}, \quad B \mapsto 10 \text{ (2)}, \quad C \mapsto 110 \text{ (3)}, \quad D \mapsto 111 \text{ (3)}.$$

The average length is

$$\bar{L} = 0.70 \cdot 1 + 0.10 \cdot 2 + 0.10 \cdot 3 + 0.10 \cdot 3 = 1.50 \text{ bits/symbol},$$

which is 0.50 bits/symbol smaller than the fixed-length 2-bit code. The more skewed the probability distribution, the larger the possible savings.

Question 2: Entropy of the 12-character set

The Shannon entropy of a discrete source with probabilities p_i is

$$H = -\sum_i p_i \log_2 p_i,$$

which measures the theoretical lower bound (in bits per symbol) for any lossless encoding on average.

We compute each term $-p_i \log_2 p_i$ and sum them. The table below lists the probabilities, code lengths (from the Fano code), the contribution $-p_i \log_2 p_i$ to the entropy, and the contribution $p_i \ell_i$ to the average length.

Symbol	p_i	$-p_i \log_2 p_i$ (bits)	code	$p_i \ell_i$ (bits)
A	0.20	0.46438562	000	$0.20 \times 3 = 0.60000000$
B	0.15	0.41054484	100	$0.15 \times 3 = 0.45000000$
C	0.12	0.36706724	010	$0.12 \times 3 = 0.36000000$
D	0.10	0.33219281	1100	$0.10 \times 4 = 0.40000000$
E	0.08	0.29150850	0110	$0.08 \times 4 = 0.32000000$
F	0.06	0.24353362	1010	$0.06 \times 4 = 0.24000000$
G	0.05	0.21609640	001	$0.05 \times 3 = 0.15000000$
H	0.05	0.21609640	1011	$0.05 \times 4 = 0.20000000$
I	0.04	0.18575425	0111	$0.04 \times 4 = 0.16000000$
J	0.03	0.15176681	1101	$0.03 \times 4 = 0.12000000$
K	0.02	0.11287712	1111	$0.02 \times 4 = 0.08000000$
L	0.10	0.33219281	1110	$0.10 \times 4 = 0.40000000$

Now sum the entropy contributions and the average-length contributions:

$$
\begin{aligned}
H &= \sum_i \Big(-p_i \log_2 p_i\Big) \\
&= 0.46438562 + 0.41054484 + 0.36706724 + 0.33219281 + 0.29150850 \\
&\quad + 0.24353362 + 0.21609640 + 0.21609640 + 0.18575425 + 0.15176681 \\
&\quad + 0.11287712 + 0.33219281 \\
&\approx 3.32401643 \text{ bits/symbol.}
\end{aligned}
$$

The sum of $p_i\ell_i$ (average code length under the Fano code) is

$$
\begin{aligned}
\bar{L} &= \sum_i p_i\ell_i \\
&= 0.6000 + 0.4500 + 0.3600 + 0.4000 + 0.3200 \\
&\quad + 0.2400 + 0.1500 + 0.2000 + 0.1600 \\
&\quad + 0.1200 + 0.0800 + 0.4000 \\
&= 3.48000000 \text{ bits/symbol.}
\end{aligned}
$$

Question 2 interpretation

The entropy $H \approx 3.3240$ bits/symbol is the *theoretical* lower bound on the average number of bits per symbol for any lossless encoding of this source (Shannon's source coding theorem). No prefix-free binary code can have average length strictly below H. In practice, a prefix code can approach H but cannot go below it; the difference $\bar{L} - H$ measures redundancy.

Question 3: Efficiency of the Fano code

We compare the Fano code's average length $\bar{L}$ to the entropy H.

$$
\begin{aligned}
\bar{L} &= 3.48000000 \text{ bits/symbol,} \\
H &= 3.32401643 \text{ bits/symbol.}
\end{aligned}
$$

The redundancy (excess average length) is

$$\bar{L} - H \approx 3.48000000 - 3.32401643 = 0.15598357 \text{ bits/symbol.}$$

The coding efficiency (fraction of the entropy attained) is

$$\eta = \frac{H}{\bar{L}} \approx \frac{3.32401643}{3.48000000} \approx 0.9552 \quad \text{(or 95.52\%).}$$

Interpretation: the Fano code uses on average 3.48 bits per symbol, while the entropy lower bound is about 3.324 bits per symbol. Thus the Fano code is quite efficient: it achieves about 95.5% of the theoretical best (or equivalently has about 0.156 bits/symbol of redundancy). For many practical purposes this is a good result; Huffman coding (an optimal prefix code for a given discrete distribution) would produce an average length that is equal to or slightly better than this Fano code (and cannot be smaller than the entropy bound).

Closing remarks (design perspective)

- When bandwidth is scarce, exploiting the non-uniformity of symbol probabilities yields immediate savings: assign short codewords to frequent symbols and longer codewords to rare symbols.
- Entropy quantifies the best average-case compression possible. Practical algorithms (Huffman, arithmetic coding) aim to approach H while maintaining constraints (prefix property, streaming decoding, etc.).
- The Fano code above is a reasonable variable-length prefix code: it is simple to construct, it respects the prefix property, and in this example it comes within a few tenths of a bit of the entropy bound.

Problem D: Word Search Puzzle

Overview

This program that creates a 10×10 word-search puzzle from a comma-separated list of words. My goal was to make something that (1) always tries to place every word, (2) allows sensible overlaps (letters must match), (3) supports all eight directions (horizontal, vertical, diagonal, both forward and backward), and (4) looks different each run by using randomness while remaining robust if randomness fails. This is a really tricky problem, as there are a lot of possible combinations of word placements!

My Initial Attempts

When I first started, I treated the puzzle as a simple grid-fill problem and tried a greedy deterministic approach: for each word, scan left-to-right, top-to-bottom and place the word at the first valid spot. That worked for tiny lists, but produced very predictable puzzles (everything clustered at the top) and often failed to place words when there were better, non-obvious placements that required different starting locations or reversed directions.

Then I tried adding reversed words explicitly (appending reversed(word) to the list) instead of supporting direction inversion. That was messy — it doubled the workload and made overlap logic confusing.

Next I added randomness: pick a direction and a random start cell; try a few times per word. This produced varied, natural-looking puzzles. But randomness alone created a non-deterministic failure mode: sometimes random picks simply never find a valid spot even when one exists. To fix that I added a deterministic fallback: if random attempts fail after a bounded number, scan every cell in every direction and place the word at the first valid spot. That hybrid approach combined the best of both worlds: varied puzzles with guaranteed placement when possible.

Design choices I committed to

- Use a fixed grid size: 10×10.
- Represent empty cells with `None` (easier logic comparisons).
- Allow all eight directions:
  (0,1), (0,-1), (1,0), (-1,0), (1,1), (-1,-1), (1,-1), (-1,1).
- If a word is contained in another word (such as THRONE in DETHRONE), remove the sub-word from the list.
- Do randomness-first (bounded attempts) then deterministic fallback.

- Limit random attempts per word to 200 to avoid infinite loops or long runs.
- After placing all words, fill remaining `None` cells with uniformly random uppercase letters A–Z.

How the final solution works (detailed, step-by-step)

I'll explain the code logic as if I'm walking through the program on a single run.

Grid initialization

- Define `size = 10`.
- Build `grid` as a list of 10 lists, each containing ten `None` values.

$$\texttt{grid} = [[\texttt{None} \text{ for } 0..\,9] \text{ for } 0..\,9].$$

- `None` is used because it cleanly differentiates unused cells from letters and prevents accidental matching with placeholder characters.

Allowed directions

I permit eight directional vectors:

$$(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (-1, -1), (1, -1), (-1, 1).$$

Each vector is a step $(\Delta r, \Delta c)$ that denotes how row and column change when you advance one letter in that direction. Negative steps allow backward placements (words written right-to-left, bottom-to-top, etc.).

The `can_place(word,row,col,dr,dc)` check

This function is the safety gate for any attempted placement. For each index i in `word` (starting at 0):

1. Compute the target cell: $r = \text{row} + dr \cdot i$, $c = \text{col} + dc \cdot i$.
2. If (r, c) is out of bounds (not in $[0, 9]$), immediately return `False` — the word would run off the grid.
3. If `grid[r][c]` is `None`, the cell is free and we can continue checking the next letter.
4. If `grid[r][c]` contains a letter that is equal to `word[i]`, treat this as a permitted overlap and continue.
5. If `grid[r][c]` contains a different letter, this placement conflicts with an existing word; return `False`.

If all letters are checked without conflict, return `True`. Because we test bounds first, we never index outside the grid.

The `place_word(word,row,col,dr,dc)` action

After `can_place` returns `True`, we write the letters into the grid: for each index i:

$$\texttt{grid[row + dr*i][col + dc*i]} \leftarrow \texttt{word[i]}.$$

No additional checks are needed because we've already validated the placement.

Per-word placement strategy

For each word in the input list, the program does:

1. **Randomized attempts:**
   - Randomly shuffle the `directions` list to avoid direction bias.
   - Repeat up to 200 times: pick a random direction (dr, dc) (one of the eight), pick a random start row and column uniformly from 0..9, and call `can_place`. If it returns `True`, call `place_word` and mark the word as placed.
   - The reason to shuffle directions and choose a random start cell each try is to produce a diverse distribution of placements and to avoid cluster artifacts.
2. **The 'deterministic' fallback:**
   - If the randomized attempts fail (after 200 tries), do a full deterministic scan: for every row r in 0..9, for every column c in 0..9, for every direction (dr, dc) in the directions list, call `can_place(r,c,dr,dc)`. Place at the first valid spot found.
   - This ensures the algorithm finds a placement if one exists, removing the brittleness of pure randomness.
3. **Failure case:**
   - If no placement is found even after the deterministic scan, raise `ValueError("Unable to place word: <word>")`. This signals to the caller that the current word set cannot fit in the 10×10 grid under the placement rules.

Finalizing the grid

After all words are successfully placed, loop through every cell; if a cell is still `None`, replace it with a random uppercase letter:

$$\texttt{grid[r][c]} \leftarrow \text{chr(random.randint(ord('}A'),\ \text{ord('}Z'))).$$

This step produces the final puzzle board ready to be printed row-by-row.

Complexity and an intuition about speed

Let W be the number of words and L be the average length of words.

- Each `can_place` run examines up to L cells (one per character).
- For randomized attempts we do at most 200 attempts per word, each attempt costing $O(L)$ checks, so that phase is $O(200 \cdot L)$ per word (which is $O(L)$ in big-O terms because 200 is constant).
- The deterministic fallback scans at most $10 \times 10 \times 8$ start-direction combinations $= 800$ placements to check; each check is $O(L)$, giving $O(800 \cdot L)$ worst-case for the fallback (again a constant times L).
- So per word worst-case is $O(\max(200, 800) \cdot L) = O(800 \cdot L)$ which is $O(L)$ with a moderate constant. For the whole input the complexity is $O(W \cdot L)$ with practical constants (here at most about 800 per word).

Because 10×10 grid and 8 directions are constants, the algorithm is effectively linear in the total number of letters attempted to place.

Edge cases, testing, and reproducibility

- **Too many or too-long words:** If the sum of all word-lengths is larger than 100, placement can still possibly succeed due to overlaps, but it's more likely to fail. In that case the deterministic fallback will eventually raise an error which is desirable — it tells you the word list is infeasible under the rules.
- **One-letter words:** They are supported naturally and act as single-cell placements; they help overlap density.
- **Case handling and input trimming:** The main block strips spaces around words; the code currently doesn't normalise case, so feeding lowercase words will place lowercase letters. In practice you probably want to `.upper()` the words before placing and optionally validate alphabetic-only input.
- **Reproducibility:** If you want reproducible puzzles, call `random.seed(some_int)` before placement. By default, the runs are non-deterministic.
- **Overlapping behavior:** Overlaps are only allowed when letters match exactly. This intentionally preserves word integrity and creates natural intersections like a crossword.

Possible improvements I considered (but did not include to keep the solution simple)

- **Scoring placements:** Instead of placing at the first deterministic spot, compute a placement score (e.g., number of matching overlaps) and choose the placement that maximizes overlap to create denser, puzzle-like intersections. This is easy to add but complicates the algorithm and removes the "first valid spot" guarantee.

- **Adaptive retry limits:** Make the random-retry limit adapt to word length or remaining free space. I used a constant 200 as a pragmatic compromise.
- **Better seeding and logging:** To debug difficult placement sets, log attempted placements (or seed the RNG so failures are reproducible).
- **Weighted letter fill:** Fill unused cells using English letter frequency instead of uniform A–Z sampling so that random letters look more word-like. That helps disguise words for players.
- **Symmetry or theme constraints:** Force symmetric layouts or align words along certain axes — useful for themed puzzles but not necessary for a general generator.

Code Implementation

Python — Word Search Puzzle

```python
import sys
import random

def create_crossword(words: list) -> list:
    """
    Generate a 10x10 word search puzzle containing the given words.

    Args:
        words: A list of words to include in the puzzle.

    Returns:
        A 2D array (list of lists) representing the word search puzzle.
    """
    # Define the grid size
    size = 10
    # Initialize the grid with None placeholders to track unused cells
    grid = [[None for _ in range(size)] for _ in range(size)]

    # Define possible placement directions as (row step, col step)
    # Horizontal, vertical, and diagonal in both forward and backward directions
    directions = [
        (0, 1),   # left to right
        (0, -1),  # right to left
        (1, 0),   # top to bottom
        (-1, 0),  # bottom to top
        (1, 1),   # diagonal down-right
        (-1, -1), # diagonal up-left
        (1, -1),  # diagonal down-left
        (-1, 1)   # diagonal up-right
    ]

    def can_place(word, row, col, dr, dc):
        """
        Check if a word can be placed starting at (row, col) with direction (dr, dc).
        Allows overlaps if characters match.
        """
        for i, ch in enumerate(word):
            r, c = row + dr * i, col + dc * i
            # Check bounds
            if not (0 <= r < size and 0 <= c < size):
                return False
            # Check overlap validity
            if grid[r][c] is not None and grid[r][c] != ch:
                return False
        return True

    def place_word(word, row, col, dr, dc):
        """
        Place a word in the grid at (row, col) moving in direction (dr, dc).
        This mutates the grid.
        """
        for i, ch in enumerate(word):
            r, c = row + dr * i, col + dc * i
            grid[r][c] = ch

    # Place each word into the grid
    for word in words:
        placed = False
        # Randomize directions and starting positions to improve distribution
        random.shuffle(directions)
        attempts = 0
        while not placed and attempts < 200:  # limit retries for runtime safety
            dr, dc = random.choice(directions)
            row = random.randint(0, size - 1)
            col = random.randint(0, size - 1)
            if can_place(word, row, col, dr, dc):
                place_word(word, row, col, dr, dc)
                placed = True
            attempts += 1
        # If still not placed after many tries, force placement linearly in first
        ↪  valid spot
        if not placed:
            for r in range(size):
                for c in range(size):
                    for dr, dc in directions:
                        if can_place(word, r, c, dr, dc):
                            place_word(word, r, c, dr, dc)
                            placed = True
                            break
                    if placed:
                        break
                if placed:
                    break
            # If absolutely no space exists, raise an error
            if not placed:
                raise ValueError(f"Unable to place word: {word}")
    # Fill empty cells with random uppercase letters for the puzzle
    for r in range(size):
        for c in range(size):
            if grid[r][c] is None:
                grid[r][c] = chr(random.randint(ord('A'), ord('Z')))

    return grid




# --- Main execution block. DO NOT MODIFY.  ---
if __name__ == "__main__":
    try:
        # Read words from first line (comma-separated)
        words_input = input().strip()
        words = [word.strip() for word in words_input.split(',')]

        # Generate the word search puzzle
        puzzle = create_crossword(words)

        # Print the result as a 2D grid
        for row in puzzle:
            print(''.join(row))

    except ValueError as e:
        print(f"Input Error: {e}", file=sys.stderr)
        sys.exit(1)
    except EOFError:
        print("Error: Not enough input lines provided.", file=sys.stderr)
        sys.exit(1)
    except Exception as e:
        print(f"An unexpected error occurred: {e}", file=sys.stderr)
        sys.exit(1)
```

Problem E: Functional Completeness of `NAND`

Statement

Show that the single binary connective `NAND` ("not-and") is functionally complete; that is, every Boolean function can be expressed using only `NAND`. Concretely, demonstrate how to obtain the usual connectives $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT) using only `NAND`.

Definition and truth table

Define $\mathrm{N}(A, B)$ to mean `NAND`(A, B), i.e. $\mathrm{N}(A, B) = \neg(A \wedge B)$. Its truth table is:

A	B	$\mathrm{N}(A, B)$
0	0	1
0	1	1
1	0	1
1	1	0

Constructing $\neg$ (NOT)

Evaluate $\mathrm{N}(A, A)$. Since $\mathrm{N}(A, A) = \neg(A \wedge A) = \neg A$, this gives logical negation.

A	$\mathrm{N}(A, A)$
0	1
1	0

$$\Rightarrow \qquad \neg A \;=\; \mathrm{N}(A, A).$$

Constructing $\wedge$ (AND)

By definition $\mathrm{N}(A, B) = \neg(A \wedge B)$. Negating this expression yields $A \wedge B$. Using the NAND-based negation from above:

$$A \wedge B \;=\; \neg\Big(\mathrm{N}(A, B)\Big) \;=\; \mathrm{N}\,\Big(\mathrm{N}(A, B),\, \mathrm{N}(A, B)\Big).$$

Truth table verification:

A	B	$\mathrm{N}(A, B)$	$\mathrm{N}(\mathrm{N}(A, B), \mathrm{N}(A, B))$
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

The last column equals the truth table of $A \wedge B$, so the identity holds.

Constructing $\vee$ (OR)

Use De Morgan's law:

$$A \vee B \;=\; \neg(\neg A \wedge \neg B).$$

Replace negations by $\mathrm{N}(\cdot,\cdot)$ applied to identical inputs:

$$\neg A = \mathrm{N}(A,A), \qquad \neg B = \mathrm{N}(B,B).$$

Then

$$A \vee B \;=\; \neg\Big(\neg A \wedge \neg B\Big) \;=\; \mathrm{N}\left(\mathrm{N}(A,A),\, \mathrm{N}(B,B)\right).$$

Truth table verification:

A	B	$(\mathrm{N}(A,A),\mathrm{N}(B,B))$	$\mathrm{N}(\mathrm{N}(A,A),\mathrm{N}(B,B))$
0	0	$(1,1)$	1
0	1	$(1,0)$	1
1	0	$(0,1)$	1
1	1	$(0,0)$	0

The last column equals the truth table of $A \vee B$, so the identity is correct.

Conclusion

We have shown that

$$\neg A = \mathrm{N}(A,A), \qquad A\wedge B = \mathrm{N}\left(\mathrm{N}(A,B),\mathrm{N}(A,B)\right), \qquad A\vee B = \mathrm{N}\left(\mathrm{N}(A,A),\mathrm{N}(B,B)\right).$$

Since the standard set $\{\neg,\wedge\}$ (or equivalently $\{\neg,\vee\}$) is functionally complete, and each of these connectives can be expressed using only N, it follows that `NAND` alone is functionally complete.